

Project 2: Understanding Cache Memories

Zhuohao Li ✉*

May 31, 2022

1 Introduction

The project2: Understanding Cache Memories is the second project for CS2503: Computer Architecture. In this project, we're gonna have a deep understanding of cache memories from a higher perspective. Besides, it's a good practice that helps us to learn more about optimized code from system perspectives. The project consists of two parts:

- **Part A: Write A Cache Simulator**

In this part, we will finish a C program to simulate the cache's behaviours and count the number of **cache hits**, **cache misses**, **evictions**. We take the *valgrind* (a memory leaking detective tools) trace as the input of this simulator, to observe the actions of the cache.

This part aims to get us better and deeper understand the cache, especially the set-associated cache and replacement strategies of cache hierarchy.

- **Part B: Optimizing Matrix Transpose**

In this part, we'll focus on how to optimize the performance of **Matrix-Transpose**, which is to store the transpose of matrix A into matrix B. To optimize the function, we need to make the cache misses as few as possible and have a full comprehension in the cache miss. This part aims to teach us the methods of optimizing a program by attaching special attention on the cache. Especially, we'll focus on **principles of locality** and get ideas about how important it is when programming.

We propose **Blocking** and several tricks to help it works. We looked deep in its theoretical optimal result and rethink how to reach that goal. This part needs more thinking.

*edith_lzh@sjtu.edu.cn | 519021911248

2 Experiments

2.1 Part A

2.1.1 Analysis

In this part, we're gonna finish `csim.c` to simply simulate cache working strategies, especially **checking if hit, replacement when missing**. Let's go!

- **Cache Mapping Strategies**

The most important in this part is fully understand how the cache works before constructing the frame of codes. In the lecture, we focused on three strategies of mapping memory address \rightarrow cache block: **fully associative, direct mapped, set associative**. Generally, they're quite similar essentially, we usually differentiate them by 3 parameters: S (**A cache is a set of 2^s cache sets**), E (**A cache set is a set of E cache lines**), B (**Each cache line stores a block, Each block has $B = 2^b$ bytes**). A memory address will be mapped to a **unique, speicific set**, and allocates to a cache line in a set by comparing **tag** and **valid code**. Totally, the cache capacity is determined by $S * B * E$. By showing Fig.1 [1] we know that they are actually all set associative, different in the set size. As shown in Fig.1, direct-mapped is 1-way set associative and fully associative is m-way set associative, where m is the number of blocks. Thus, what we care about is how to implement set associative. Fig.2 shows the abstract of transporting memory address to cache block. In this lab, we **don't care about offset**, what we need to do is simulating placement and replacement strategies. Simply **count hits, misses, and evictions**

- **Hints**

In this part, the cache is just 2D array of cache lines:

- `struct cache_line cache[S][E]`
- $S=2^s$, is the number of sets
- E is associativity

Each cache line has: Valid bit, Tag, LRU counter

- **Cache Replacement**

In this lab, we utilize **LRU** to replace missed cache block. LRU is – Least Recently Used replacement policy. It evicts the least recently used block from the cache to make room for the next block. We could simply use a **queue** or **Time Stamps** to implement this. Alg.1 shows the basic idea of LRU used in this lab. [1]

- **Parse *valgrind* File**

Generated by *valgrind*, a open source memory leaking detective tools will play a role in inserting files as the input of `csim.c`. To sparse the

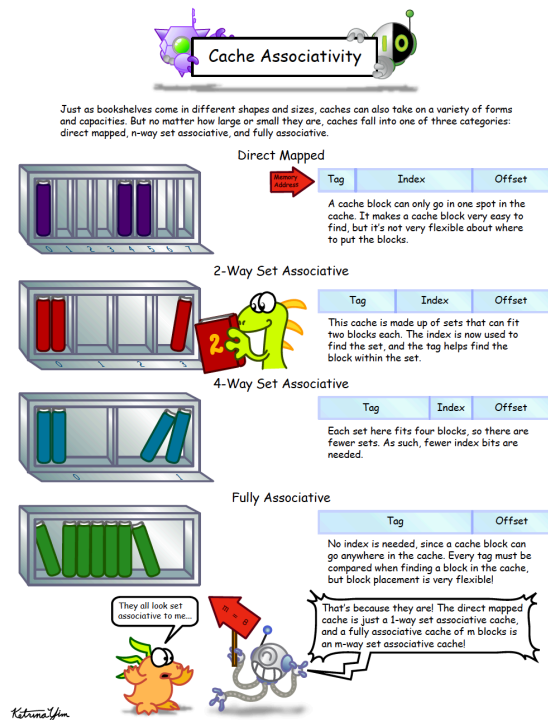


Figure 1: Interesting Illustration of Set-Associated Cache

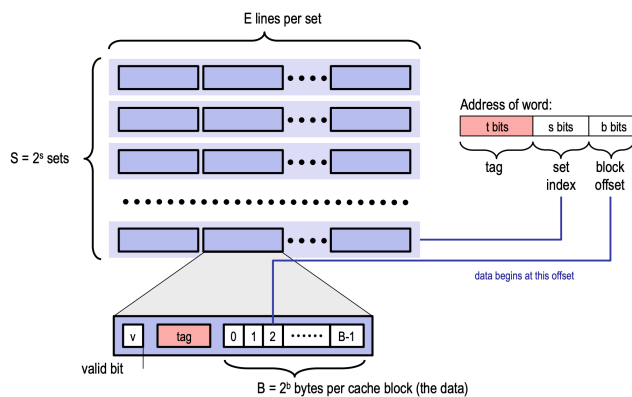


Figure 2: Abstract of Mapping

UNIX command line and file manipulations, please refer to `getopy()` and `fscanf()` in assignment. I'm gonna discuss the `trace` file, which is

Algorithm 1: LRU Replacement

input : Blocks Set $B = \{b_1, b_2, \dots, b_n\}$

output: Replaced Block b

```
1 Define  $t_{min} = \infty$ 
2 Define  $ir = 0$ 
3 for each  $b_i$  in  $B$  do
4   if  $b_i.t \leq t_{min}$  then
5      $t_{min} = b_i.t$ ;
6      $ir = i$ ;
7 return  $b_{ir}$ 
```

used to test our cache and compared to `csim_ref`. Generally, there're 4 operations here in `trace`.

- **I. Instruction Load**. Just ignore it in our lab because we only focus on data load/store.
- **L. Data Load**. It may cause: 1) hit (data in cache) 2) miss (data not in cache) 3) miss eviction (conflicts occur when load data from main memory to cache)
- **S. Data Store**. It may cause: 1) hit (data in cache) 2) miss (data not in cache) 3) miss eviction (conflicts occur when load data from main memory to cache). Exactly same as **L**
- **M. Data Modify**. It may cause: 1) 2 hits (data in cache) 2) miss + hit (data not in cache) 3) miss eviction + hit (conflicts occur when load data from main memory to cache). **M** is just a **S** after **L**. Pls kindly understand this because it's very vital next.

2.1.2 Code

I'm gonna attach some core code only in this section, for details, please refer to my source code uploaded.

First of all, I defined a **basic data structure** for cache line.

```
1 // primary data structure
2 typedef struct {
3   int valid; // valid bit, 0 is invalid, 1 is valid
4   int tag;   // tag bit
5   int time;  // LRU time, manipulate LRU
6 } cache_line;
7 // abstract of cache
8 /** *cache specifies each sets, **cache specifies each block
9     */
9 cache_line** cache;
```

Remember that **S** for sets, **E** for ways, **b** for blocks. To manipulate each cache block, I defined `index_s`, `index_E`, `index_b`. They're all set to the **global variables**. Take a look at the following code:

```
1 // parameters definition
2 int S;           // num of sets
3 int E;           // num of ways == lines per set
4 int B;           // block size
5 int index_s;     // store the index of sets when allocates
6 int index_E;     // store the index of lines in a set.
7 int index_b;     // store the index of blocks in memory
8 int current_time; // manipulate LRU replacement
9 char input_file[100]; // input file generated by others
10 FILE* file_pointer; // manipulate input_file
```

The main function of the code is divided into 3 main parts:

1. parse the command
2. preparation work, including legal check, global variable initialization
3. function call including `init_cache()`, `operation_parse()`, `free_cache()`, `fclose(file_pointer)`, `printSummary(hits, misses, evictions)`. We'll talk about them later.

For the **parse command section**, referred to materials from CMU, it mainly works with `getopt()` as follows:

```
1 // parse command line
2 while ((params = getopt(argc, argv, "s:E:b:t:hv")) != -1) {
3     switch (params) {
4         // just list a few, the complete version refers to the
4         // source code pls
5         case 's':
6             index_s = atoi(optarg);
7             break;
8         case 'E':
9             index_E = atoi(optarg);
10            break;
11        case 'b':
12            index_b = atoi(optarg);
13            break;
14    }
15 }
```

For **preparation work**, we're gonna check if all `index` is correct(at least ≥ 0) and examines whether files in `./trace` are work. **S**, **B**, **E** could be derived

from 64-bit address in this lab, just by << is enough. That's why we utilize power of 2 for convenience.

For **function calls**, here're the APIs I'm gonna use later:

```

1 // let's begin
2 init_cache();                // initialize cache
3 operation_parse();           // parse instructions in
    input file
4 free_cache();                // free cache real memory
5 fclose(file_pointer);        // free file memory
6 printSummary(hits, misses, evictions); // report result

```

- `init_cache()`:

We will use `malloc` to allocate memory for a specific cache object. Due to 2-level pointer I used here for cache, cache itself has a capacity of `S` and each `cache[i]` has a capacity of `E` in general. We initialize all of them with `valid = 0`, `-1` for tag and time.

```

1 void init_cache() {
2     // malloc for a new cache
3     /* for a cache at the top, it has a memory size of S
4     cache = (cache_line**)malloc(sizeof(cache_line*) * S)
5     ;
6     // clear all zeros
7     for (int i = 0; i < S; ++i) {
8         /* for each set, it has a memory size of E
9         cache[i] = (cache_line*)malloc(sizeof(cache_line)
10             * E);
11         for (int j = 0; j < E; ++j) {
12             cache[i][j].valid = 0;
13             cache[i][j].tag = -1;
14             cache[i][j].time = -1;
15         }
16     }
17 }

```

- `operation_parse()`

From the `./trace` file, we need to use `fscanf` (recommended by assignment) to sparse the instruction. **I, L, S, M** are the 4 cases we care about. It could usually be implemented by a `switch` to sparse. For **I**, as mentioned before, we just `continue` it. And for **L** or **S**, we call `update_cache(address)` to do place and replace. Note that for **M**, as we treated it as **S** after **L** we should maintain the order of each case in the `switch`, which means no `break` should be there to correct its executing order. [2]

```

1 void operation_parse() {
2     unsigned int address; // address, 2nd argument
3     int size; // size, the number of bytes accessed by
4                 the operation, 3rd
5     // argument
6     char operation;
7     char command; // manipulate, -v, -t, -h
8     while (fscanf(file_pointer, "%c %xu%c%u ", &
9                 operation, &address, &command,
10                &size) > 0) {
11         switch (operation) {
12             case 'I': // I: just continue
13                 continue;
14             case 'L': // L
15                 update_cache(address);
16                 break;
17             case 'M': // M
18                 update_cache(address);
19             case 'S': // S
20                 update_cache(address);
21                 break;
22             default:
23                 break;
24         }
25         update_time(); // plus each cache line's time 1 in
26                         every visit
27     }
28 }

```

- `free_cache()`

It's a simple loop for a 2-level pointer to free memory allocation

- `update_cache(unsigned int address)`

This is the most vital part in partA. The logic is simple, which is

1. Parse the instruction we got before to make it clear about which is exactly the corresponding set number of tag value. Set number will be stored in `set_spec`, tag value will be stored in `tag_spec`. As the 64-bit address is conducted as `tag set offset`, I used a AND and right shift operation to get that.
2. search in the cache set by comparing `tag` and check `valid`
 - If hit, update and return.
 - If not hit, go to next step.

3. Search in the set for the empty space to replace.
 If found, update and return.
 If not found, go to next step
4. Use **LRU** to replace the block. Update and return. The **LRU** is implemented by a **time stamp**, algorithm logic is shown before in Alg.1

```

1 void update_cache(unsigned int address) {
2     // sparse tag value
3     // address: |tag|set|offset|
4     int tag_spec =
5     (address >> (index_b + index_s)); // address right
        shift for
6     // index_b+index_s sparse set value
7     int set_spec =
8     (address >> index_b) &
9     ((0xFFFFFFFF) >> (64 - index_s)); // it should be |
        set|, bit-wise AND
10    // could sparse |set| from origin
11    // look up in a set
12    for (int i = 0; i < E; ++i) {
13        /* hit: tag is matched && valid == 1 */
14        if ((cache[set_spec][i].tag == tag_spec) && cache[
15            set_spec][i].valid == 1) {
16            hits++; // hit plus 1
17            cache[set_spec][i].time =
18                current_time; // update the visit time to do
                LRU
19            return;
20        }
21    }
22    for (int i = 0; i < E; ++i) {
23        // invalid --- miss
24        if (cache[set_spec][i].valid == 0) {
25            cache[set_spec][i].valid = 1; // turn to valid
26            cache[set_spec][i].tag = tag_spec;
27            cache[set_spec][i].time = current_time;
28            misses++;
29            return;
30        }
31    }
32    // valid but not match --- eviction
33    evictions++;
34    misses++;
35    //***** LRU replacement *****/

```



```

35     int min_time = 10000; // simulate \infty
36     int ir;                // replaced block
37     for (int i = 0; i < E; ++i) {
38         if (cache[set_spec][i].time < min_time) {
39             min_time = cache[set_spec][i].time; // update
               time
40             ir = i;                               // update ir
41             /* code */
42         }
43     }
44     // after LRU, ir is what we're gonna replace
45     cache[set_spec][ir].tag = tag_spec;
46     cache[set_spec][ir].time = current_time;
47 }

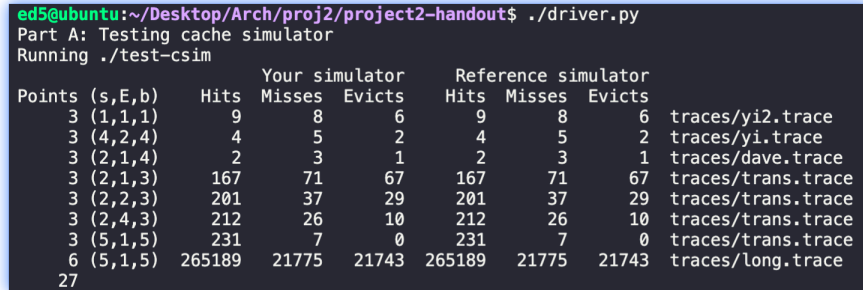
```

- update_time() is just doing plus 1 each time

man_help(), -h, -v and -t is ignored here. They're not important and used for debugging and using exclusively.

2.1.3 Evaluation

>>make clean and make in the directory, then ./python.py by Python2.x interpreter. The result is shown as Fig.3 and reached full mark of 27.



```

ed5@ubuntu:~/Desktop/Arch/proj2/project2-handout$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim

```

Points (s,E,b)	Your simulator			Reference simulator			
	Hits	Misses	Evicts	Hits	Misses	Evicts	
3 (1,1,1)	9	8	6	9	8	6	traces/yi2.trace
3 (4,2,4)	4	5	2	4	5	2	traces/yi.trace
3 (2,1,4)	2	3	1	2	3	1	traces/dave.trace
3 (2,1,3)	167	71	67	167	71	67	traces/trans.trace
3 (2,2,3)	201	37	29	201	37	29	traces/trans.trace
3 (2,4,3)	212	26	10	212	26	10	traces/trans.trace
3 (5,1,5)	231	7	0	231	7	0	traces/trans.trace
6 (5,1,5)	265189	21775	21743	265189	21775	21743	traces/long.trace
27							

Figure 3: test in PartA

2.2 Part B

2.2.1 Techniques

In this part, we're gonna implement an optimized transpose_submit(int M, int N, int A[N][M], int B[M][N]) in trans.c to transpose a fixed size

matrix. The reasons why we need to deal with it is **principles of locality**, both **tempory** and **spatial**. As a matrix is **2-dimension** in this lab and the store strategy of it is **row-first**, we need to use specific tricks to reduce misses. The main technique here is we called, **Blocking**.

Blocking can significantly improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called *blocks*. To implement this we usually introduces **stride** which is very common in **machine learing** or **parallelisim computing** research fields. In fact, a very popular simulator, **zSim** [4] proposes by MIT and Stanford back to 2013 has utilized some of the ideas to reality.

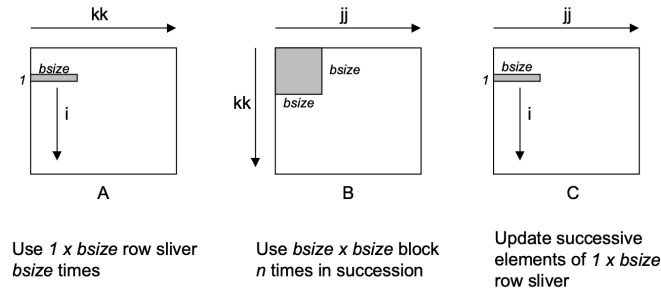


Figure 4: **Graphical interpretation of blocked matrix multiply:** The innermost (j, k) loop pair multiplies a $1 \times bsize$ sliver of A by a $bsize \times bsize$ block of B and accumulates into a $1 \times bsize$ sliver of C

The classic example of this is **matrix multiplication** shown in Fig.4. We also utilized blocking in **matrix transpose**. They're quite similar.

2.2.2 Analysis

In this part, we're gonna use cache placement strategies of $s=5$, $E=1$, $b=5$, which means that there're totally $2^5 = 32$ sets, and using direct-mapping ($E = 1$) way. Block size is $2^5 = 32$ (contains 8 **int**)too. So, for a single cache set, we could store 8 **int** at a time. For a single cache, we could store 32×8 **int** at a time.

The testbench is buit on three different sizes of matrixes: 32×32 , 64×64 , 61×67 . We'll seperately optimize them.

2.2.3 Code

32×32

A cache line could contain 8 **int**, in the original 32×32 matrix, we have 32 integers per row, which can be contained by 4 cache line. A cache could contain 8 rows, so we divide the 32×32 matrix into 8×8 block chunks. In a inner-chunk, we did transposition. (line \rightarrow column).

Note that if the data is on the main diagonal, we won't transpose immediately. If so, we could reduce the 2 misses happened when transposition at diagonal. I declared `tmp` to store the value and `index` to store its position. After the other data has been transposed, we could directly assign the 8 diagonal value at a time. The code is:

```

1 int i, j, tmp, index;
2 int row_Block, col_Block; // top row and column manipulation
   with stride
3
4 if (M == 32) { // divide the the 32X32 block into 8X8 ,
   decrease the number
5   // of misses
6   // stride is 8, for 8X4=32, each
7   for (row_Block = 0; row_Block < N; row_Block += 8) {
8     for (col_Block = 0; col_Block < M; col_Block += 8) {
9       for (i = row_Block; i < row_Block + 8; i++) {
10        for (j = col_Block; j < col_Block + 8; j++) {
11          if (i != j) {
12            B[j][i] = A[i][j]; // transpose
13          } else {
14            tmp = A[i][j]; // i==j means the data is
                           on the diagonal. if we
15            // set B right now ,the misses and
16            // evictions will increase . because
17            // the cache set of B is same to A.
18            index = i;}}
19          if (col_Block == row_Block) { // just set B on
               the diagonal. other
20            // than shouldn't set the B
21            B[index][index] = tmp; }}}}

```

The theoretical optimal result is **256** misses. To save pages, I'm not gonna talk about why it is. Maybe I'll work on that by coding in the future.

64 × 64

The part is much more difficult than the other 2, basically, we're gonna divide it into 64 **8 × 8** blocks. But here're some tricks.

The first trick is we call **Read All Only Once** that shows from code line10 → 17. To be more specific, when doing blocking, we read the full line of the block once. For example, the matrix *A* has the size 32×32 . When CPU reads *A*[0][0], one cache line is filled with *A*[0][0], *A*[0][1], ..., *A*[0][7] because row-first. If we just read one element like *A*[0][6] then transpose the element and get the value of *B*[6][0], the cache line is replaced with *B*[0][0], ..., *B*[0][7]. Thus, there are extra elements that are not read but placed in the cache. If we not use them in this time, they are still in requirement. We can read these elements at one time.

The second trick we call is **Save and Load**. [3]. To be specific, in 64×64

matrix, the cache can store 4 rows of matrix. But if we perform *blocking* of 8×8 , it can only be filled with *line1* to *line4* or *line5* to *line8*. What we're gonna do is dividing it into **more fine-grained**. As mentioned before, we first read 4 elements in A's lower left quarter and store the transposition of them in B's lower left quarter. This is called *Save*. Then, when transposing A's upper right corner, we first *load* the elements in B we store before and do the transposition of the upper right corner and the lower left corner of B.

```

1 // using 4 + 8 = 12 variable, following coding style
2 int t, t1, t2, t3, t4, t5, t6, t7, t8;
3 for (row_Block = 0; row_Block < N; row_Block += 8)
4 for (col_Block = 0; col_Block < M; col_Block += 8) // 8X8
    blocking
5 { // inner is blocking 4X4
6   // save and load
7   for (t = row_Block; t < row_Block + 4; ++t) {
8     // read all only once
9     t1 = A[t][col_Block];
10    t2 = A[t][col_Block + 1];
11    t3 = A[t][col_Block + 2];
12    t4 = A[t][col_Block + 3];
13    t5 = A[t][col_Block + 4];
14    t6 = A[t][col_Block + 5];
15    t7 = A[t][col_Block + 6];
16    t8 = A[t][col_Block + 7];
17
18    B[col_Block][t] = t1;
19    B[col_Block + 1][t] = t2;
20    B[col_Block + 2][t] = t3;
21    B[col_Block + 3][t] = t4;
22    B[col_Block][t + 4] = t5;
23    B[col_Block + 1][t + 4] = t6;
24    B[col_Block + 2][t + 4] = t7;
25    B[col_Block + 3][t + 4] = t8;
26  }
27  for (t = col_Block; t < col_Block + 4; ++t) {
28    t1 = A[row_Block + 4][t];
29    t2 = A[row_Block + 5][t];
30    t3 = A[row_Block + 6][t];
31    t4 = A[row_Block + 7][t];
32    t5 = B[t][row_Block + 4];
33    t6 = B[t][row_Block + 5];
34    t7 = B[t][row_Block + 6];
35    t8 = B[t][row_Block + 7];
36
37    B[t][row_Block + 4] = t1;

```

```

38         B[t][row_Block + 5] = t2;
39         B[t][row_Block + 6] = t3;
40         B[t][row_Block + 7] = t4;
41         B[t + 4][row_Block] = t5;
42         B[t + 4][row_Block + 1] = t6;
43         B[t + 4][row_Block + 2] = t7;
44         B[t + 4][row_Block + 3] = t8;
45     }
46     for (t = row_Block + 4; t < row_Block + 8; ++t) {
47         t1 = A[t][col_Block + 4];
48         t2 = A[t][col_Block + 5];
49         t3 = A[t][col_Block + 6];
50         t4 = A[t][col_Block + 7];
51         B[col_Block + 4][t] = t1;
52         B[col_Block + 5][t] = t2;
53         B[col_Block + 6][t] = t3;
54         B[col_Block + 7][t] = t4; }}

```

The theoretical optimal result is **1024** misses.

61×67

Totally like 32×32 , we made a chunk of block, 16×16 , and here's the code:

```

1 // separate the the 61X67 block into 16X16 , decrease the
  // number of misses
2 for (row_Block = 0; row_Block < N; row_Block += 16) {
3     for (col_Block = 0; col_Block < M; col_Block += 16) {
4         for (i = row_Block; i < row_Block + 16 && (i < N); i++)
5             {
6                 for (j = col_Block; j < col_Block + 16 && (j < M);
7                     j++) {
8                     if (i != j) {
9                         B[j][i] = A[i][j];
10                    } else {
11                        tmp = A[i][j]; // i==j means is the diagonal
12                                     // if we
13                                     // set B right now ,the misses and
14                                     // evictions will increase . because
15                                     // the cache set of B is same to A.
16                        index = i; }}
17                if (col_Block == row_Block) { // just set B on the
18                    diagonal. other
19                    // than shouldn't set the B
20                    B[index][index] = tmp; }}}

```

2.2.4 Evaluation

```
./test-trans -M 32 -N 32
```

hits	misses	evictions
1766	287	255

```
./test-trans -M 64 -N 64
```

hits	misses	evictions
9066	1179	1147

```
./test-trans -M 61 -N 67
```

hits	misses	evictions
6197	1985	1953

By `linux >> ./driver.py`, we could get the complete evaluation result at a time. The result is shown in Fig.5 and luckily I **reached full mark of 26**.

Combined patA and partB, I got the full mark of 53!

```
Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
           Points  Max pts  Misses
Csim correctness    27.0     27
Trans perf 32x32      8.0      8      287
Trans perf 64x64      8.0      8     1179
Trans perf 61x67     10.0     10     1985
      Total points    53.0     53
```

Figure 5: test for PartB

3 Conclusion

3.1 Problems

1. In Part A, building the structure of the simulator from zero is quite hard to understand. It takes some effort to further understand and make it clear what we should do by dividing the stages into **1) placement 2)re-placement**.

2. In Part B, it's hard to make it clear how the ***Blocking*** works because code with *blocking* is difficult to write and read. However, I scanned materials from CMU and got to know how to optimize the function.
3. It's hard to get the full score when constructing 64×64 matrix compared to the others. I've read many articles and looked deep into the theory behind behaviors with endless efforts and tries. And it works.
4. It's a little bit hard to understand how SOTA of performing, but it is actually very interesting.

3.2 Achievements

- Successfully build a cache simulation `csim.c`, with the same output as `csim-ref`.
- Use *Blocking* with 2 other tricks in the guidance of principles of locality to optimize 3 different matrix sizes tests.
- Construct a number of testbenches to compare different techniques.
- Write all the codes with verbose comments. Follow the coding style correctly. Make it readable and easy to understand.
- Write code and instruction basically on what we're truly understand and try to build a software and hardware co-design method to implement that kind of stuff.

References

- [1] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.
- [2] lab lecture CMU. <http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/recitations/rec07.pdf>.
- [3] <https://zhuanlan.zhihu.com/p/387662272>. Zhihu: optimal resolution in cachelab.
- [4] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.